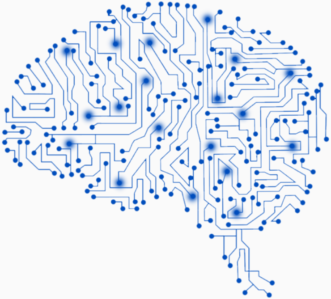
Master 1 Statistique & Data Science, Ingénierie Mathématique

# Apprentissage pour l'image
# Machine learning for image processing

## Course II – Introduction to Artificial Neural Networks: Backpropagation

Emile Pierret

Jeudi 16 mars 2023



# Objectives

At the end of the course :

- Know what is a CNN (Convolutional Neural Network)
- Implement the training of a CNN for classification with Pytorch

For this session :

- Backpropagation to compute gradients in Neural Networks

# A simple example

Let consider $f : (x, y) \in \mathbb{R}^2 \mapsto \log(xy)$. How to differentiate $f$ step by step with respect to $x$ and $y$ ?

# A simple example

Let consider $f : (x, y) \in \mathbb{R}^2 \mapsto \log(xy)$. How to differentiate $f$ step by step with respect to $x$ and $y$ ? We can split $f$ such that :

1. $v = xy$

## A simple example

Let consider $f : (x, y) \in \mathbb{R}^2 \mapsto \log(xy)$. How to differentiate $f$ step by step with respect to $x$ and $y$ ? We can split $f$ such that :

1. $v = xy$
2. $w = \log(v) = \log(xy)$

# A simple example

Let consider $f : (x, y) \in \mathbb{R}^2 \mapsto \log(xy)$. How to differentiate $f$ step by step with respect to $x$ and $y$ ? We can split $f$ such that :

1. $v = xy$
2. $w = \log(v) = \log(xy)$
3. $z = w$

# A simple example

Let consider $f : (x, y) \in \mathbb{R}^2 \mapsto \log(xy)$. How to differentiate $f$ step by step with respect to $x$ and $y$ ? We can split $f$ such that :

1. $v = xy$
2. $w = \log(v) = \log(xy)$
3. $z = w$

As a reminder, for $g, h : \mathbb{R} \to \mathbb{R}$,

$$(g \circ f)'(x) = g'(f(x))f'(x)$$

# A simple example

Let consider $f : (x, y) \in \mathbb{R}^2 \mapsto \log(xy)$. How to differentiate $f$ step by step with respect to $x$ and $y$ ? We can split $f$ such that :

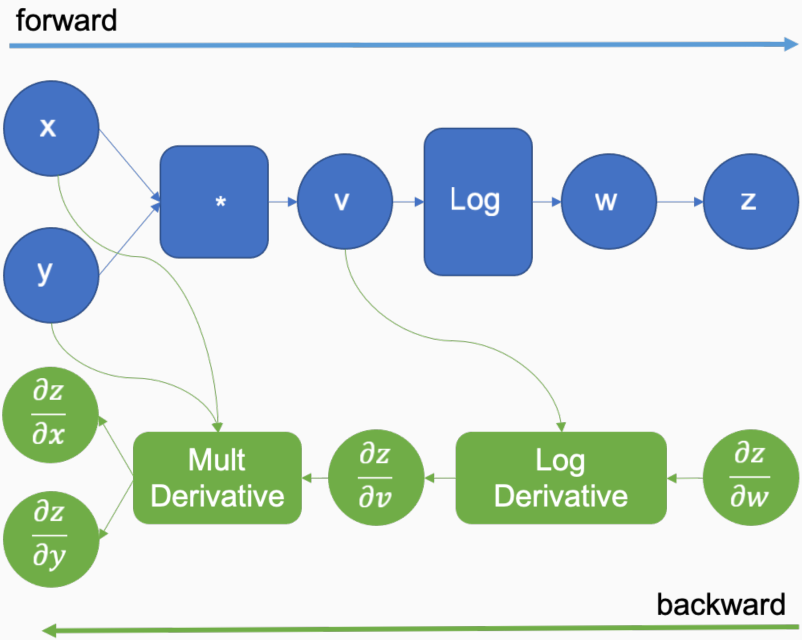
1. $v = xy$
2. $w = \log(v) = \log(xy)$
3. $z = w$

As a reminder, for $g, h : \mathbb{R} \to \mathbb{R}$,

$$(g \circ f)'(x) = g'(f(x))f'(x)$$

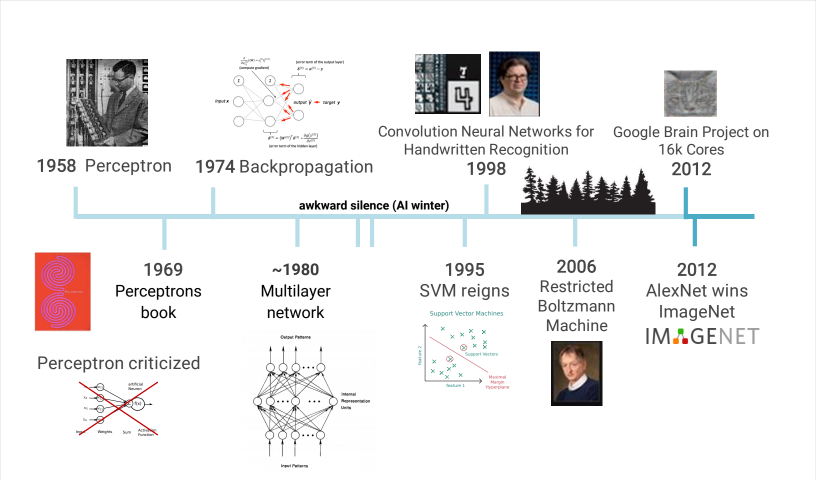
Consequently,

$$\begin{aligned}
\frac{\partial z}{\partial x} &= \frac{\partial z}{\partial w}\frac{\partial w}{\partial x} \\
&= \frac{\partial z}{\partial w}\frac{\partial w}{\partial v}\frac{\partial v}{\partial x} \\
&= 1 \times \frac{1}{v} \times y \\
&= \frac{1}{x}
\end{aligned}$$


forward
x
y
*
v
Log
w
z
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
Mult Derivative
$\frac{\partial z}{\partial v}$
Log Derivative
$\frac{\partial z}{\partial w}$
backward

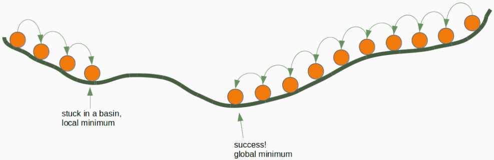
# Machine learning – Timeline

## Timeline of (deep) learning

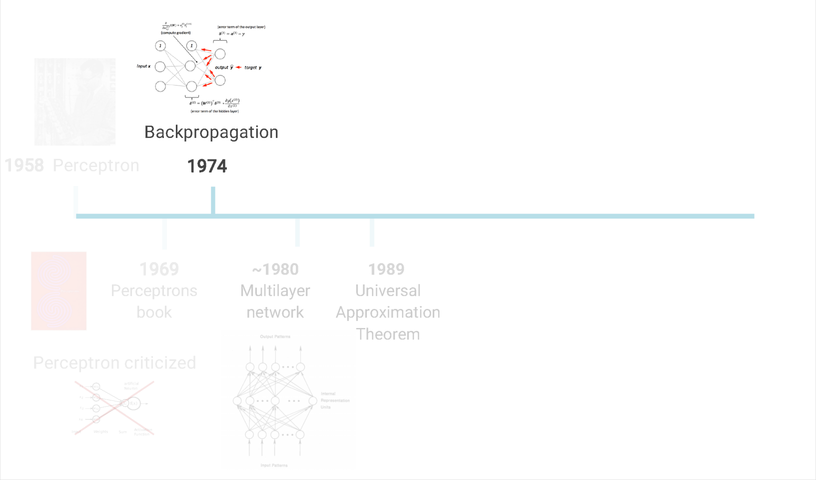



(Source: Lucas Masuch & Vincent Lepetit)

# Backpropagation

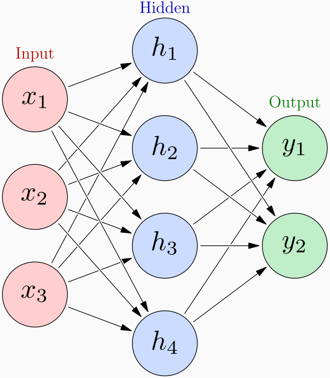


## Learning with backpropagation




(Source: *Lucas Masuch & Vincent Lepetit*)

## Artificial neural network / Multilayer perceptron / NeuralNet



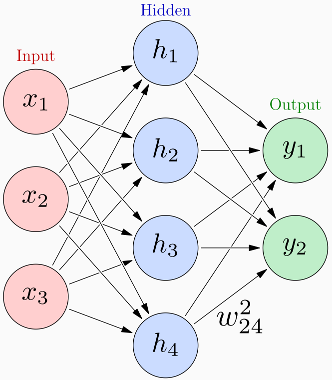

$$h_1 = g_1\left(w^1_{11}x_1 + w^1_{12}x_2 + w^1_{13}x_3 + b^1_1\right)$$
$$h_2 = g_1\left(w^1_{21}x_1 + w^1_{22}x_2 + w^1_{23}x_3 + b^1_2\right)$$
$$h_3 = g_1\left(w^1_{31}x_1 + w^1_{32}x_2 + w^1_{33}x_3 + b^1_3\right)$$
$$h_4 = g_1\left(w^1_{41}x_1 + w^1_{42}x_2 + w^1_{43}x_3 + b^1_4\right)$$

$$y_1 = g_2\left(w^2_{11}h_1 + w^2_{12}h_2 + w^2_{13}h_3 + w^2_{14}h_4 + b^2_1\right)$$
$$y_2 = g_2\left(w^2_{21}h_1 + w^2_{22}h_2 + w^2_{23}h_3 + w^2_{24}h_4 + b^2_2\right)$$

$w^k_{ij}$ synaptic weight between previous node $j$ and next node $i$ at layer $k$.

$g_k$ are any activation function applied to each coefficient of its input vector.

## Artificial neural network / Multilayer perceptron / NeuralNet



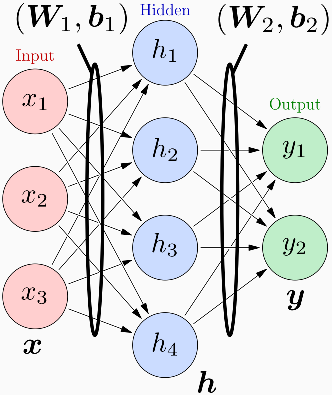

$$h_1 = g_1\left(w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3 + b_1^1\right)$$
$$h_2 = g_1\left(w_{21}^1 x_1 + w_{22}^1 x_2 + w_{23}^1 x_3 + b_2^1\right)$$
$$h_3 = g_1\left(w_{31}^1 x_1 + w_{32}^1 x_2 + w_{33}^1 x_3 + b_3^1\right)$$
$$h_4 = g_1\left(w_{41}^1 x_1 + w_{42}^1 x_2 + w_{43}^1 x_3 + b_4^1\right)$$

$$y_1 = g_2\left(w_{11}^2 h_1 + w_{12}^2 h_2 + w_{13}^2 h_3 + w_{14}^2 h_4 + b_1^2\right)$$
$$y_2 = g_2\left(w_{21}^2 h_1 + w_{22}^2 h_2 + w_{23}^2 h_3 + w_{24}^2 h_4 + b_2^2\right)$$

$w_{ij}^k$ synaptic weight between previous node $j$ and next node $i$ at layer $k$.

$g_k$ are any activation function applied to each coefficient of its input vector.

## Artificial neural network / Multilayer perceptron / NeuralNet




$$h_1 = g_1\left(w^1_{11}x_1 + w^1_{12}x_2 + w^1_{13}x_3 + b^1_1\right)$$
$$h_2 = g_1\left(w^1_{21}x_1 + w^1_{22}x_2 + w^1_{23}x_3 + b^1_2\right)$$
$$h_3 = g_1\left(w^1_{31}x_1 + w^1_{32}x_2 + w^1_{33}x_3 + b^1_3\right)$$
$$h_4 = g_1\left(w^1_{41}x_1 + w^1_{42}x_2 + w^1_{43}x_3 + b^1_4\right)$$
$$\boldsymbol{h} = g_1\left(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1\right)$$

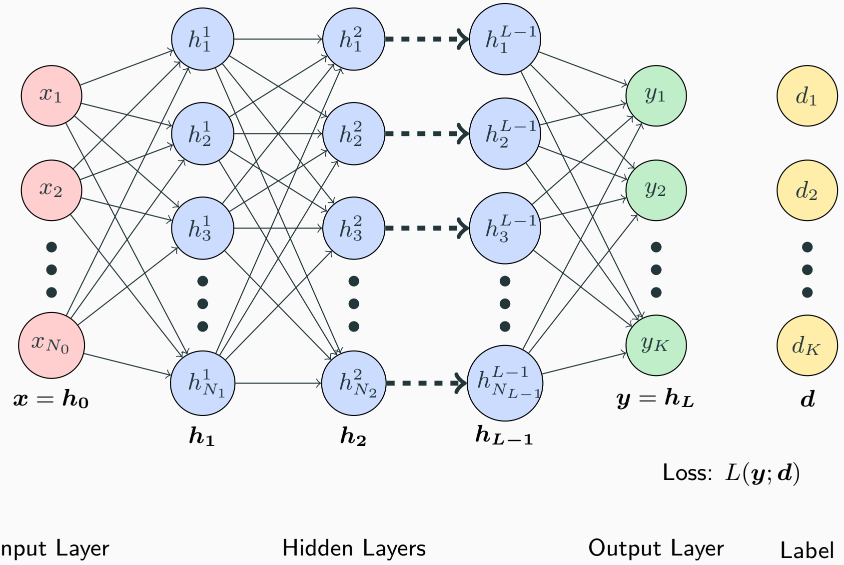
$$y_1 = g_2\left(w^2_{11}h_1 + w^2_{12}h_2 + w^2_{13}h_3 + w^2_{14}h_4 + b^2_1\right)$$
$$y_2 = g_2\left(w^2_{21}h_1 + w^2_{22}h_2 + w^2_{23}h_3 + w^2_{24}h_4 + b^2_2\right)$$
$$\boldsymbol{y} = g_2\left(\boldsymbol{W}_2\boldsymbol{h} + \boldsymbol{b}_2\right)$$

$w^k_{ij}$ synaptic weight between previous node $j$ and next node $i$ at layer $k$.

$g_k$ are any activation function applied to each coefficient of its input vector.

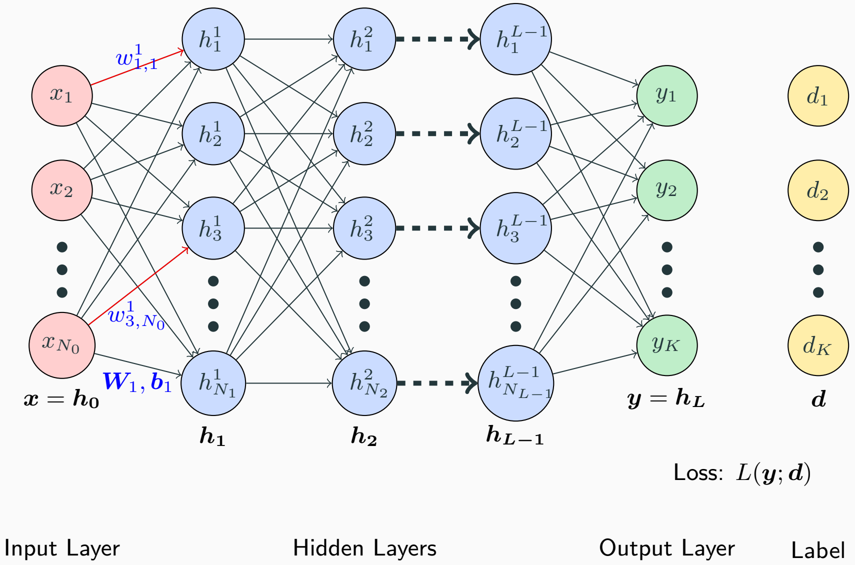
The matrices $\boldsymbol{W_k}$ and biases $\boldsymbol{b}_k$ are learned from labeled training data.

**Recall the feedforward structure**



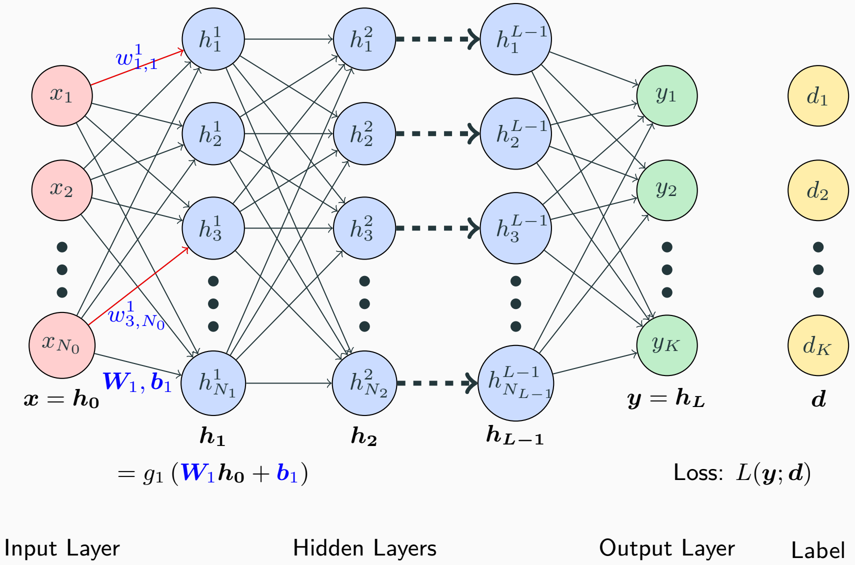
# Feedforward Artificial Neural Network

**Recall the feedforward structure**



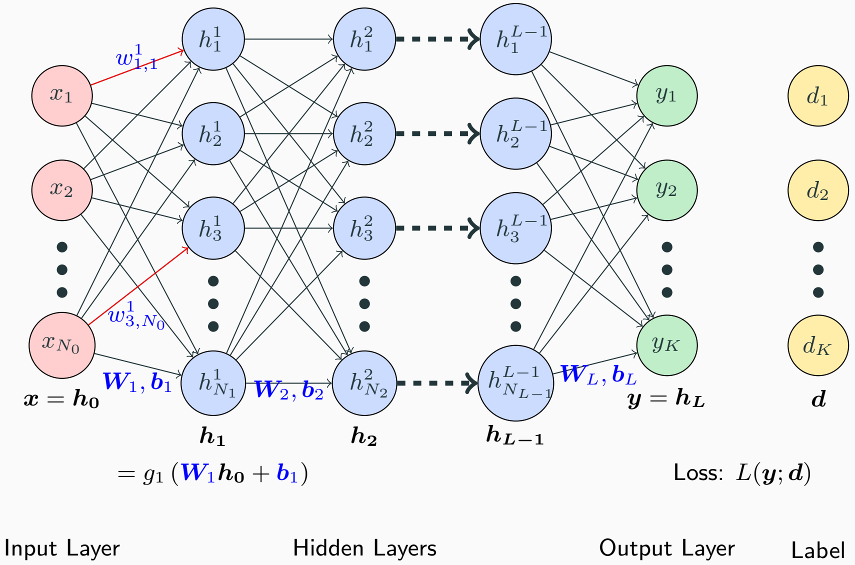
# Feedforward Artificial Neural Network

## Recall the feedforward structure

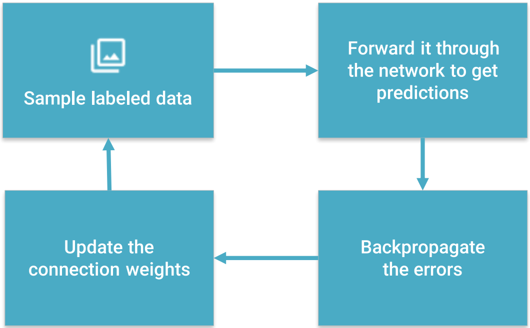


# Feedforward Artificial Neural Network

**Recall the feedforward structure**



## Training process




Learns by generating an error signal that measures the difference between the predictions of the network and the desired values and then using this error signal to change the weights (or parameters) so that predictions get more accurate.

(*Source: Lucas Masuch*)

# ANN – Optimization

- The parameters of the neural network are

$$\boldsymbol{W} = (\boldsymbol{W}_1, \boldsymbol{b}_1, \boldsymbol{W}_2, \boldsymbol{b}_2, \ldots, \boldsymbol{W}_L, \boldsymbol{b}_L)$$

# ANN – Optimization

- The parameters of the neural network are

$$\boldsymbol{W} = (\boldsymbol{W}_1, \boldsymbol{b}_1, \boldsymbol{W}_2, \boldsymbol{b}_2, \ldots, \boldsymbol{W}_L, \boldsymbol{b}_L)$$

- Training the network = minimizing the training loss $E(\boldsymbol{W})$

**Objective:** $\min_{\boldsymbol{W}} E(\boldsymbol{W})$ where $E(W) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{T}} L(\boldsymbol{y}^i; \boldsymbol{d}^i)$

$$\Rightarrow \quad \nabla E(\boldsymbol{W}) = \begin{pmatrix} \frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{W}_1} & \frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{b}_1} & \ldots & \frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{W}_L} & \frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{b}_L} \end{pmatrix}^T = 0$$

- **Solution:** no closed-form solutions

# ANN – Optimization

- The parameters of the neural network are

$$\boldsymbol{W} = (\boldsymbol{W}_1, \boldsymbol{b}_1, \boldsymbol{W}_2, \boldsymbol{b}_2, \ldots, \boldsymbol{W}_L, \boldsymbol{b}_L)$$

- Training the network = minimizing the training loss $E(\boldsymbol{W})$

**Objective:** $\min_{\boldsymbol{W}} E(\boldsymbol{W})$ where $E(W) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{T}} L(\boldsymbol{y}^i; \boldsymbol{d}^i)$

$$\Rightarrow \quad \nabla E(\boldsymbol{W}) = \begin{pmatrix} \frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{W}_1} & \frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{b}_1} & \ldots & \frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{W}_L} & \frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{b}_L} \end{pmatrix}^T = 0$$

- **Solution:** no closed-form solutions $\Rightarrow$ use (stochastic) gradient descent.

# ANN – Optimization

- The parameters of the neural network are

$$\boldsymbol{W} = (\boldsymbol{W}_1, \boldsymbol{b}_1, \boldsymbol{W}_2, \boldsymbol{b}_2, \ldots, \boldsymbol{W}_L, \boldsymbol{b}_L)$$

- Training the network = minimizing the training loss $E(\boldsymbol{W})$

**Objective:** $\min_{\boldsymbol{W}} E(\boldsymbol{W})$ where $E(W) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{T}} L(\boldsymbol{y}^i; \boldsymbol{d}^i)$

$$\Rightarrow \quad \nabla E(\boldsymbol{W}) = \begin{pmatrix} \frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{W}_1} & \frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{b}_1} & \ldots & \frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{W}_L} & \frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{b}_L} \end{pmatrix}^T = 0$$

- **Solution:** no closed-form solutions $\Rightarrow$ use (stochastic) gradient descent.
- $\frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{W}_k}$ not really rigorous, we will use the notation

$$\nabla_{\boldsymbol{W}_k} E(\boldsymbol{W}) \quad \text{and} \quad \nabla_{\boldsymbol{b}_k} E(\boldsymbol{W}).$$

# ANN – Optimization

## Minimizing training loss

For multilayer neural networks $\boldsymbol{W} \mapsto E(\boldsymbol{W})$ is non-convex

$\Rightarrow$ No guarantee of convergence.

Even if convergence occurs, the solution depends on the initialization and the step size/learning rate $\gamma$.

Nevertheless, really good minima or saddle points are reached in practice by

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \gamma \nabla E(\boldsymbol{W}^t), \quad \gamma > 0$$

Gradient descent can be expressed coordinate by coordinate as:

$$w_{i,j}^{k,t+1} \leftarrow w_{i,j}^{k,t} - \gamma \frac{\partial E(\boldsymbol{W}^t)}{\partial w_{i,j}^k}$$

for all weights $w_{i,j}^k$ linking a node $j$ to a node $i$ in the next layer $k$.

$\Rightarrow$ The algorithm to compute $\frac{\partial E(\boldsymbol{W})}{\partial w_{i,j}^k}$ for ANNs is called **backpropagation**.

# ANN – Optimization

- In practice we only use **stochastic** gradient descent with batch of training set.
- The complete loss is :

$$E(W) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{T}} L(\boldsymbol{y}^i; \boldsymbol{d}^i)$$

- For some random small subset (e.g. batch) $\mathcal{S} \subset \mathcal{T}$, consider

$$E(\boldsymbol{W}; \mathcal{S}) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{S}} L(\boldsymbol{y}^i; \boldsymbol{d}^i)$$

- Our **goal** is to compute the gradient

$$\nabla_{\boldsymbol{W}_k} E(\boldsymbol{W}; \mathcal{S}) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{S}} \nabla_{\boldsymbol{W}_k} L(\boldsymbol{y}^i; \boldsymbol{d}^i).$$

# ANN – Optimization

- In practice we only use **stochastic** gradient descent with batch of training set.
- The complete loss is :

$$E(W) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{T}} L(\boldsymbol{y}^i; \boldsymbol{d}^i)$$

- For some random small subset (e.g. batch) $\mathcal{S} \subset \mathcal{T}$, consider

$$E(\boldsymbol{W}; \mathcal{S}) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{S}} L(\boldsymbol{y}^i; \boldsymbol{d}^i)$$

- Our **goal** is to compute the gradient

$$\nabla_{\boldsymbol{W}_k} E(\boldsymbol{W}; \mathcal{S}) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{S}} \nabla_{\boldsymbol{W}_k} L(\boldsymbol{y}^i; \boldsymbol{d}^i).$$

- Why is this relevant to minimize $E(\boldsymbol{W}) = E(\boldsymbol{W}; \mathcal{T})$ ?

## ANN – Optimization

- **Stochastic** gradient descent: For some random small subset (e.g. batch) $\mathcal{S} \subset \mathcal{T}$, our **goal** is to compute the noisy gradient

$$\nabla_{\boldsymbol{W}_k} E(\boldsymbol{W}; \mathcal{S}) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{S}} \nabla_{\boldsymbol{W}_k} L(\boldsymbol{y}^i; \boldsymbol{d}^i).$$

- **Unbiased approximation:** As soon as $\mathcal{S}$ spans uniformly the whole training set $\mathcal{T}$,

$$\begin{aligned}
\mathbb{E}_{\mathcal{S}} \left( \nabla_{\boldsymbol{W}_k} E(\boldsymbol{W}; \mathcal{S}) \right) &= \mathbb{E}_{\mathcal{S}} \left( \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{S}} \nabla_{\boldsymbol{W}_k} L(\boldsymbol{y}^i; \boldsymbol{d}^i) \right) \\
&= \mathbb{E}_{\mathcal{S}} \left( \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{T}} \mathbf{1}_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{S}} \nabla_{\boldsymbol{W}_k} L(\boldsymbol{y}^i; \boldsymbol{d}^i) \right) \\
&= \frac{|\mathcal{S}|}{|\mathcal{T}|} \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{T}} \nabla_{\boldsymbol{W}_k} L(\boldsymbol{y}^i; \boldsymbol{d}^i) = \frac{|\mathcal{S}|}{|\mathcal{T}|} \nabla_{\boldsymbol{W}_k} E(\boldsymbol{W}).
\end{aligned}$$

- Conclusion: In expectation the noisy gradient is equal to the gradient using the whole training dataset (unbiased estimator).

# ANN – Optimization

**Loss functions:** Classical loss functions are:

**For regression:** $\boldsymbol{d}^i \in \mathbb{R}^K$

- Square error

$$E(\boldsymbol{W}) = \sum_{(\boldsymbol{x}^i,\boldsymbol{d}^i)\in\mathcal{T}} \frac{1}{2}\|\boldsymbol{y}^i - \boldsymbol{d}^i\|_2^2 = \sum_{(\boldsymbol{x}^i,\boldsymbol{d}^i)\in\mathcal{T}} \frac{1}{2}\sum_k (y_k^i - d_k^i)^2$$

**For multi-class classification:** $d^i \in \{1,\ldots,K\}$, coded by $\boldsymbol{d}^i \in \{0,1\}^K$,

- Cross-entropy with softmax as the last layer

$$E(\boldsymbol{W}) = -\sum_{(\boldsymbol{x}^i,\boldsymbol{d}^i)} \sum_{k=1}^{K} d_k^i \log y_k^i \quad \text{with} \quad \boldsymbol{y}^i = f(\boldsymbol{x}^i;\boldsymbol{W}) = \mathrm{softmax}(\boldsymbol{a}^i) \in (0,1)^K.$$

- Cross-entropy with softmax included in loss (PyTorch convention): $\boldsymbol{y}^i = \boldsymbol{a}^i$ is the output of the last linear layer:

$$E(\boldsymbol{W}) = -\sum_{(\boldsymbol{x}^i,\boldsymbol{d}^i)} \left[ a_{d^i} - \log\left(\sum_{k=1}^{K} \exp(a_k)\right)\right] \quad \text{with } d^i \text{ the class of } \boldsymbol{x}^i.$$

# ANN – Optimization

- The loss functions are of the form

$$E(\boldsymbol{W}) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i)} L(\boldsymbol{y}^i; \boldsymbol{d}^i)$$

- By linearity,

$$\nabla E(\boldsymbol{W}) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i)} \nabla L(\boldsymbol{y}^i; \boldsymbol{d}^i)$$

- There the neural net output $\boldsymbol{y}^i = f(\boldsymbol{x}^i; \boldsymbol{W})$ is a function of the input data $\boldsymbol{x}^i$ and the neural weights $\boldsymbol{W}$.
- We know the gradient of $L(\boldsymbol{y}^i; \boldsymbol{d}^i)$ with respect to the variable $\boldsymbol{y}$
    - Regression/Square error:

$$L(\boldsymbol{y}; \boldsymbol{d}) = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{d}\|_2^2 \quad \Rightarrow \quad \nabla_{\boldsymbol{y}} L(\boldsymbol{y}; \boldsymbol{d}) = \boldsymbol{y} - \boldsymbol{d}$$

    - Multi-class classification/cross-entropy:

$$L(\boldsymbol{y}; \boldsymbol{d}) = -y_d + \log\left(\sum_{k=1}^{K} \exp(y_k)\right) \Rightarrow (\nabla_{\boldsymbol{y}} L(\boldsymbol{y}; \boldsymbol{d}))_\ell = \mathrm{softmax}(\boldsymbol{y})_\ell - \delta_{\ell,d}.$$

# ANN – Optimization

- The loss functions are of the form

$$E(\boldsymbol{W}) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i)} L(\boldsymbol{y}^i; \boldsymbol{d}^i)$$

- By linearity,

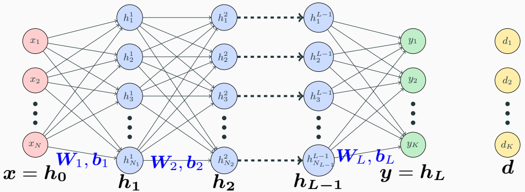
$$\nabla E(\boldsymbol{W}) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i)} \nabla L(\boldsymbol{y}^i; \boldsymbol{d}^i)$$

- There the neural net output $\boldsymbol{y}^i = f(\boldsymbol{x}^i; \boldsymbol{W})$ is a function of the input data $\boldsymbol{x}^i$ and the neural weights $\boldsymbol{W}$.
- We know the gradient of $L(\boldsymbol{y}^i; \boldsymbol{d}^i)$ with respect to the variable $\boldsymbol{y}$
- We still need to compute

$$\nabla_{\boldsymbol{W}_k} L(\boldsymbol{y}; \boldsymbol{d}) \quad \text{and} \quad \nabla_{\boldsymbol{b}_k} L(\boldsymbol{y}; \boldsymbol{d}) \quad \text{for } k = 0, \dots, L.$$

- For simplicity above we will use the notation $E = L(\boldsymbol{y}; \boldsymbol{d})$, that is considering only one point.

# ANN – Backpropagation




Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

## Forward pass

Initialization:

$\boldsymbol{h}_0 = \boldsymbol{x}$

**for** layer $k = 1$ **to** $L$ **do**

Linear unit:

$\boldsymbol{a}_k = \boldsymbol{W}_k \boldsymbol{h}_{k-1} + \boldsymbol{b}_k$

Componentwise non-linear activation:

$\boldsymbol{h}_k = g_k(\boldsymbol{a}_k)$

**end**

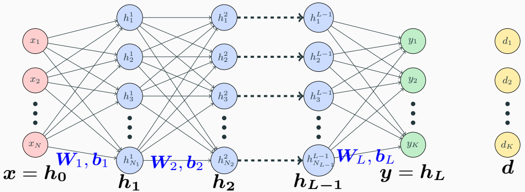
Output layer:

$\boldsymbol{y} = \boldsymbol{h}_L$

Compute loss:

$E = L(\boldsymbol{y}; \boldsymbol{d})$

# ANN – Backpropagation




Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

## Forward pass

Initialization:

$\boldsymbol{h}_0 = \boldsymbol{x}$

**for** layer $k = 1$ **to** $L$ **do**

Linear unit:

$\boldsymbol{a}_k = \boldsymbol{W}_k \boldsymbol{h}_{k-1} + \boldsymbol{b}_k$

Componentwise non-linear activation:

$\boldsymbol{h}_k = g_k(\boldsymbol{a}_k)$

**end**

Output layer:

$\boldsymbol{y} = \boldsymbol{h}_L$

Compute loss:

$E = L(\boldsymbol{y}; \boldsymbol{d})$

## Backward pass

**Goal:** Compute the gradient with respect to all parameters

$$\frac{\partial E}{\partial w_{i,j}^k} = ? \qquad \frac{\partial E}{\partial b_i^k} = ?$$

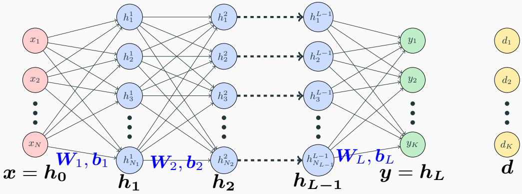
for all

$k \in \{1, \ldots, L\}$,

$i \in \{1, \ldots, N_k\}$,

$j \in \{1, \ldots, N_{k-1}\}$.

# ANN – Backpropagation



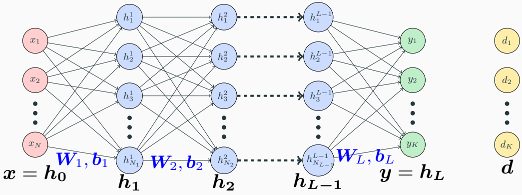

Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

**Going backward**

- We know how to compute the loss function and its gradient:

$$\nabla_{\boldsymbol{h}_L} E = \nabla L(\boldsymbol{y}; \boldsymbol{d})$$

# ANN – Backpropagation




Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

**Gradient with respect to last linear unit output $\boldsymbol{a}_L$**

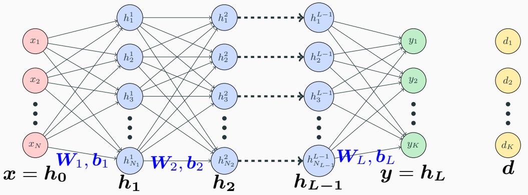
$$\boldsymbol{h}_L = g_L(\boldsymbol{a}_L)$$

That is for all $i \in \{1, \ldots, N_L\}$, $h_i^L = g_L(a_i^L)$. By the chain rule,

$$\frac{\partial E}{\partial a_i^L} = \frac{\partial E}{\partial h_i^L} \frac{\partial h_i^L}{\partial a_i^L} = [\nabla_{\boldsymbol{h}_L} E]_i \, g_L'(a_i^L)$$

**Vector formula:** $\nabla_{\boldsymbol{a}_L} E = \nabla_{\boldsymbol{h}_L} E \odot g_L'(\boldsymbol{a}_L)$

where $\odot$ is the componentwise product between vectors, ie Hadamard product.




Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

## Gradient with respect to bias of last linear unit $b_L$

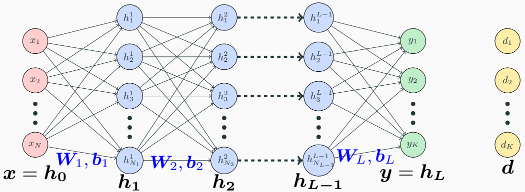
$$\boldsymbol{a}_L = \boldsymbol{W}_L \boldsymbol{h}_{L-1} + \boldsymbol{b}_L$$

That is for all $i \in \{1, \ldots, N_L\}$, $a_i^L = \sum_{j=1}^{N_{L-1}} w_{i,j}^L h_j^{L-1} + b_i^L$.

By the chain rule, for all $i \in \{1, \ldots, N_L\}$,

$$\frac{\partial E}{\partial b_i^L} = \frac{\partial E}{\partial a_i^L} \underbrace{\frac{\partial a_i^L}{\partial b_i^L}}_{=1} = \frac{\partial E}{\partial a_i^L} = [\nabla_{\boldsymbol{a}_L} E]_i$$

**Vector formula:** $\nabla_{\boldsymbol{b}_L} E = \nabla_{\boldsymbol{a}_L} E$




Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

**Gradient with respect to weights of last linear unit $W_L$**

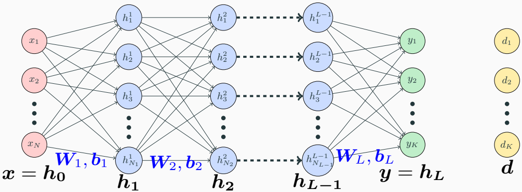
$$\boldsymbol{a}_L = \boldsymbol{W}_L \boldsymbol{h}_{L-1} + \boldsymbol{b}_L$$

That is for all $i \in \{1, \dots, N_L\}$, $a_i^L = \sum_{j=1}^{N_{L-1}} w_{i,j}^L h_j^{L-1} + b_i^L$.

By the chain rule, for all $i \in \{1, \dots, N_L\}$ and $j \in \{1, \dots, N_{L-1}\}$

$$\frac{\partial E}{\partial w_{i,j}^L} = \frac{\partial E}{\partial a_i^L} \underbrace{\frac{\partial a_i^L}{\partial w_{i,j}^L}}_{=h_j^{L-1}} = \frac{\partial E}{\partial a_i^L} h_j^{L-1} = \left[\nabla_{\boldsymbol{a}_L} E\right]_i \left[\boldsymbol{h}_{L-1}\right]_j$$

**Matrix formula:** $\nabla_{\boldsymbol{W}_L} E = \nabla_{\boldsymbol{a}_L} E \, \boldsymbol{h}_{L-1}^T$

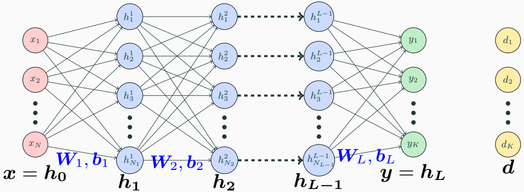



Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

## Gradients for last layer parameters

Given the gradient with respect to the output layer $\nabla_{\boldsymbol{h}_L} E$, so far we can compute:

- $\nabla_{\boldsymbol{a}_L} E = \nabla_{\boldsymbol{h}_L} E \odot g'_L(\boldsymbol{a}_L)$
- $\nabla_{\boldsymbol{b}_L} E = \nabla_{\boldsymbol{a}_L} E$
- $\nabla_{\boldsymbol{W}_L} E = \nabla_{\boldsymbol{a}_L} E \, \boldsymbol{h}_{L-1}^T$



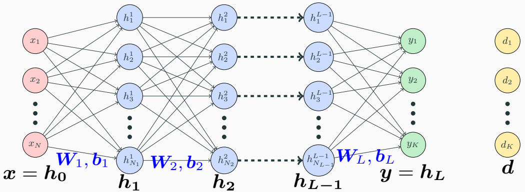

Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

## Gradients for last layer parameters

Given the gradient with respect to the output layer $\nabla_{\boldsymbol{h}_L} E$, so far we can compute:

- $\nabla_{\boldsymbol{a}_L} E = \nabla_{\boldsymbol{h}_L} E \odot g'_L(\boldsymbol{a}_L)$
- $\nabla_{\boldsymbol{b}_L} E = \nabla_{\boldsymbol{a}_L} E$
- $\nabla_{\boldsymbol{W}_L} E = \nabla_{\boldsymbol{a}_L} E \, \boldsymbol{h}_{L-1}^T$

**How can we compute the gradients for the parameters of layer $L-1$?**




Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

## Gradients for last layer parameters

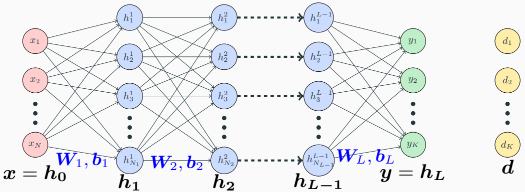
Given the gradient with respect to the output layer $\nabla_{\boldsymbol{h}_L} E$, so far we can compute:

- $\nabla_{\boldsymbol{a}_L} E = \nabla_{\boldsymbol{h}_L} E \odot g'_L(\boldsymbol{a}_L)$
- $\nabla_{\boldsymbol{b}_L} E = \nabla_{\boldsymbol{a}_L} E$
- $\nabla_{\boldsymbol{W}_L} E = \nabla_{\boldsymbol{a}_L} E \, \boldsymbol{h}_{L-1}^T$

**How can we compute the gradients for the parameters of layer $L-1$?**

We need the expression of the gradient with respect to the last but one hidden layer $\boldsymbol{h}_{L-1}$... and then the same formulas apply!

$$\nabla_{\boldsymbol{h}_{L-1}} E = ?$$



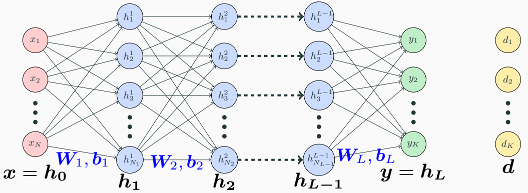

Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

## Gradient with respect to the last but one hidden layer $\boldsymbol{h}_{L-1}$

Here, even to compute the scalar partial derivative $\dfrac{\partial E}{\partial h_j^{L-1}}$, we need to use differential calculus for multivariate functions since $h_j^{L-1}$ appears in each component of $\boldsymbol{a}_L$:

For all $i \in \{1, \ldots, N_L\}$, $a_i^L = \displaystyle\sum_{j=1}^{N_{L-1}} w_{i,j}^L h_j^{L-1} + b_i^L$.

# ANN – Backpropagation




Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

## Gradient with respect to the last but one hidden layer $\boldsymbol{h}_{L-1}$

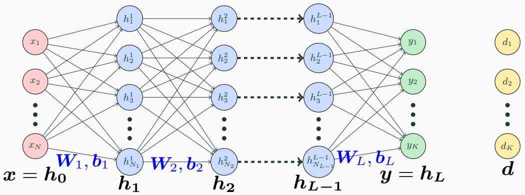
Let us recall the derivative rule for composition with affine maps:

$$\text{For} \quad \varphi(\boldsymbol{x}) = f(\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}) \quad \text{one has} \quad \nabla\varphi(\boldsymbol{x}) = \boldsymbol{A}^T \nabla f(\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}).$$

Using the decomposition

$$\begin{array}{ccccc} \mathbb{R}^{N_{L-1}} & \to & \mathbb{R}^{N_L} & \to & \mathbb{R} \\ \boldsymbol{h}_{L-1} & \mapsto & \boldsymbol{a}_L = \boldsymbol{W}_L \boldsymbol{h}_{L-1} + \boldsymbol{b}_L & \mapsto & E \end{array}$$

**Vector formula:** $\nabla_{\boldsymbol{h}_{L-1}} E = {\boldsymbol{W}_L}^T \nabla_{\boldsymbol{a}_L} E$

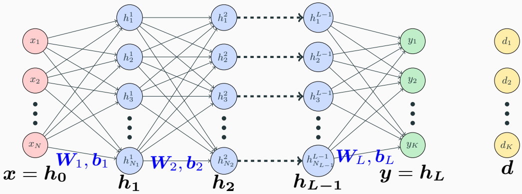



Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

We know $\nabla_{\boldsymbol{h}_{L-1}} E$ and :

- $\boldsymbol{h}_{L-1} = g_{L-1}(\boldsymbol{a}_{L-1}) \Rightarrow \nabla_{\boldsymbol{a}_{L-1}} E = \nabla_{\boldsymbol{h}_{L-1}} E \odot g'_{L-1}(\boldsymbol{a}_{L-1})$
- $\boldsymbol{a}_{L-1} = W_{L-1}\boldsymbol{h}_{L-2} + \boldsymbol{b}_{L-1}$
- As before,
- $\nabla_{\boldsymbol{b}_{L-1}} E = \nabla_{\boldsymbol{a}_{L-1}} E$
- $\nabla_{\boldsymbol{W}_{L-1}} E = \nabla_{\boldsymbol{a}_{L-1}} E \boldsymbol{h}^T_{L-2}$

# ANN – Backpropagation



Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

## Forward pass

Initialization:

$\boldsymbol{h}_0 = \boldsymbol{x}$

**for** layer $k = 1$ **to** $L$ **do**

Linear unit:

$\boldsymbol{a}_k = \boldsymbol{W}_k \boldsymbol{h}_{k-1} + \boldsymbol{b}_k$

Componentwise non-linear activation:

$\boldsymbol{h}_k = g_k(\boldsymbol{a}_k)$

**end**

Output layer:

$\boldsymbol{y} = \boldsymbol{h}_L$

Compute loss:

$E = L(\boldsymbol{y}; \boldsymbol{d})$

## Backward pass

Initialization: Gradient of output layer:

$\nabla_{\boldsymbol{h}_L} E = \nabla L(\boldsymbol{y}; \boldsymbol{d})$

**for** layer $k = L$ **to** $1$ **do**

Componentwise gain of error:

$\boldsymbol{\delta}_k = \nabla_{\boldsymbol{a}_k} E = \nabla_{\boldsymbol{h}_k} E \odot g'_k(\boldsymbol{a}_k)$

Gradient of layer bias:

$\nabla_{\boldsymbol{b}_k} E = \boldsymbol{\delta}_k$

Gradient of weights:

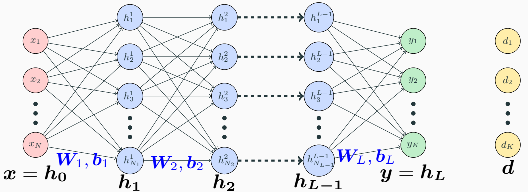
$\nabla_{\boldsymbol{W}_k} E = \boldsymbol{\delta}_k \boldsymbol{h}_{k-1}^T$

Gradient of previous hidden layer:

$\nabla_{\boldsymbol{h}_{k-1}} E = \boldsymbol{W}_k^T \boldsymbol{\delta}_k$

**end**

# ANN – Backpropagation




Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

## Forward pass

Initialization:

$\boldsymbol{h}_0 = \boldsymbol{x}$

**for** layer $k = 1$ **to** $L$ **do**

Linear unit:

$\boldsymbol{a}_k = \boldsymbol{W}_k \boldsymbol{h}_{k-1} + \boldsymbol{b}_k$ **(stored)**

Componentwise non-linear activation:

$\boldsymbol{h}_k = g_k(\boldsymbol{a}_k)$ **(stored)**

**end**

Output layer:

$\boldsymbol{y} = \boldsymbol{h}_L$

Compute loss:

$E = L(\boldsymbol{y}; \boldsymbol{d})$

## Backward pass

Initialization: Gradient of output layer:

$\nabla_{\boldsymbol{h}_L} E = \nabla L(\boldsymbol{y}; \boldsymbol{d})$

**for** layer $k = L$ **to** $1$ **do**

Componentwise gain of error:

$\boldsymbol{\delta}_k = \nabla_{\boldsymbol{a}_k} E = \nabla_{\boldsymbol{h}_k} E \odot g_k'(\boldsymbol{a}_k)$

Gradient of layer bias:

$\nabla_{\boldsymbol{b}_k} E = \boldsymbol{\delta}_k$

Gradient of weights:

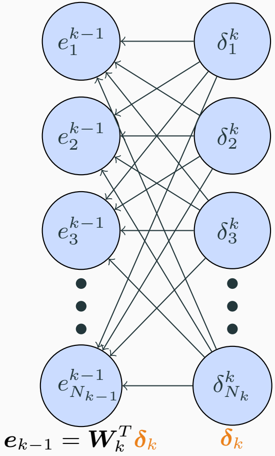
$\nabla_{\boldsymbol{W}_k} E = \boldsymbol{\delta}_k \boldsymbol{h}_{k-1}^T$

Gradient of previous hidden layer:

$\nabla_{\boldsymbol{h}_{k-1}} E = \boldsymbol{W}_k^T \boldsymbol{\delta}_k$

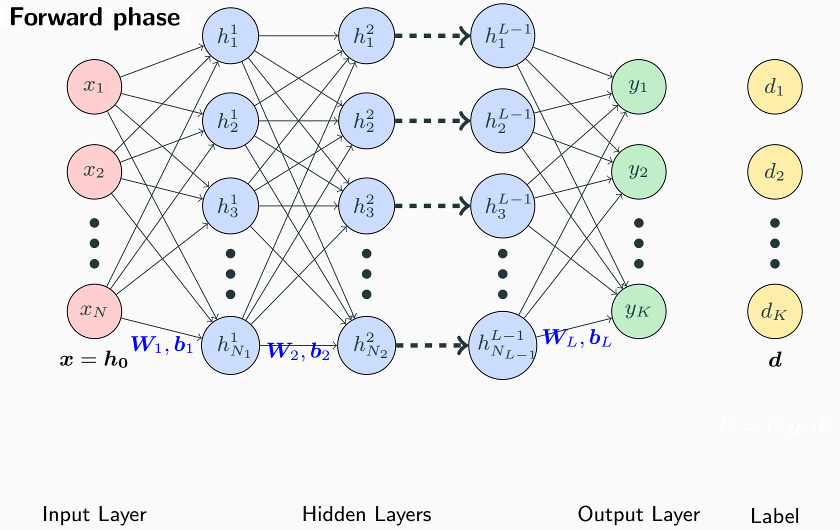
**end**

## Error backpropagation



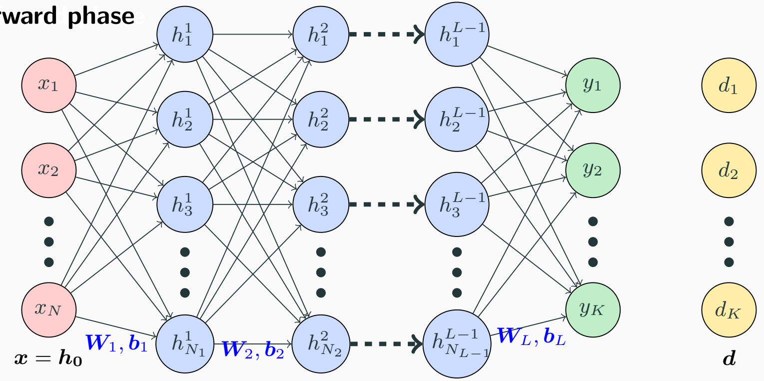

- Gradient of previous hidden layer: $\boldsymbol{e}_{k-1} = \nabla_{\boldsymbol{h}_{k-1}} E = \boldsymbol{W}_k^T \boldsymbol{\delta}_k$
- Multiplying by $\boldsymbol{W}_k^T$ corresponds to passing to the linear layer in reverse order.
- The error is backpropagated layer by layer to compute the gradient with respect to each layer parameters.

## Error backpropagation



## Error backpropagation

**Forward phase**

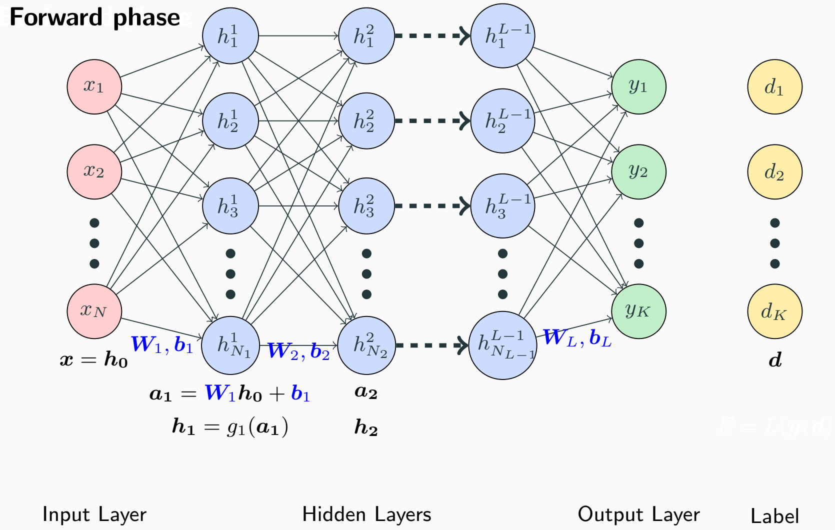



$$\boldsymbol{a_1} = \boldsymbol{W}_1 \boldsymbol{h_0} + \boldsymbol{b}_1$$

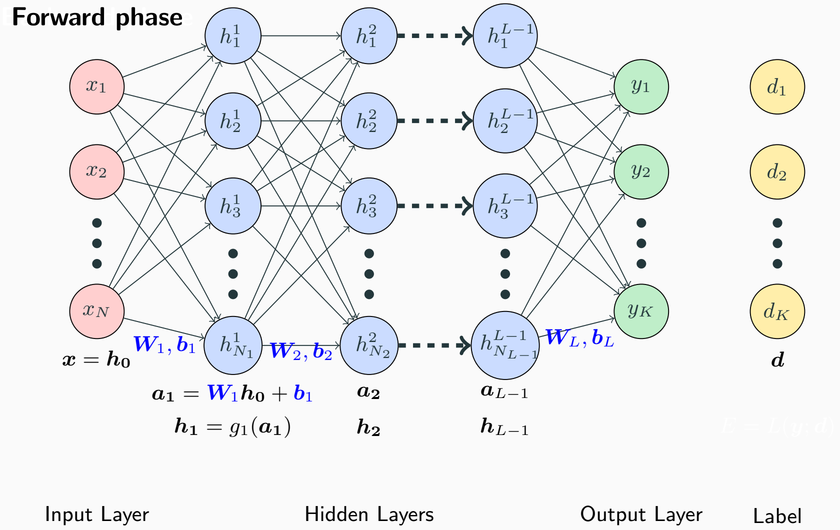
$$\boldsymbol{h_1} = g_1(\boldsymbol{a_1})$$

Input Layer | Hidden Layers | Output Layer | Label

## Error backpropagation

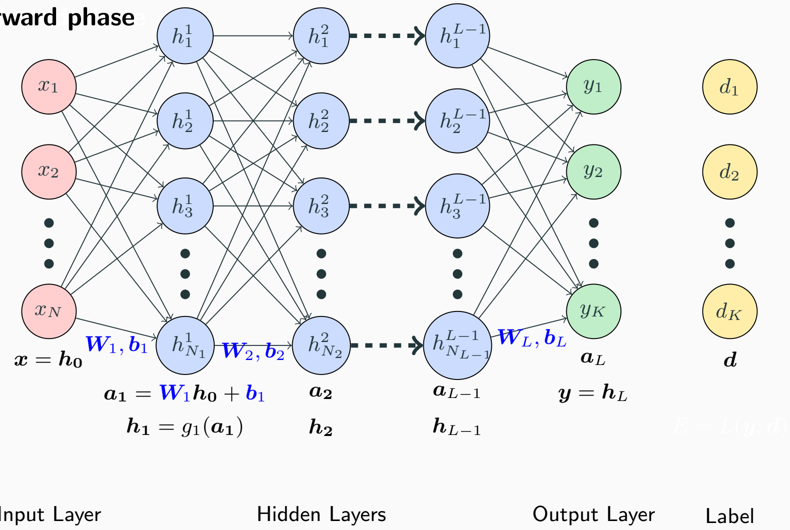


## Error backpropagation

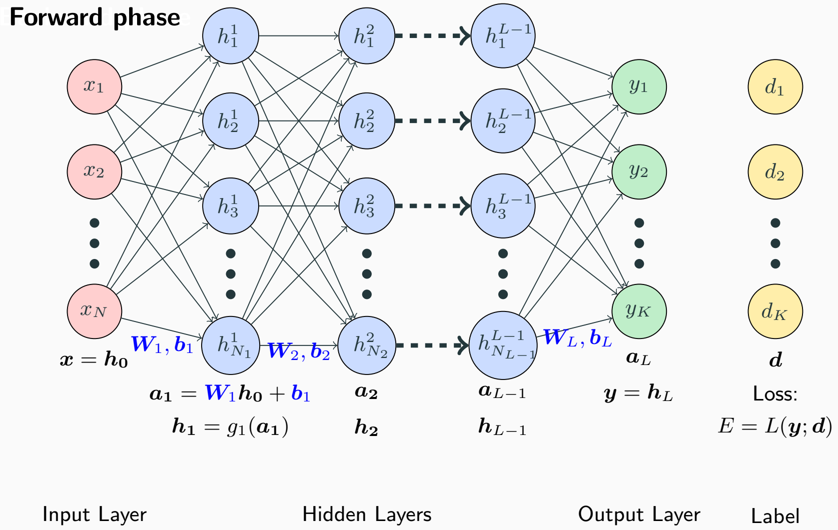


## Error backpropagation

**Forward phase**

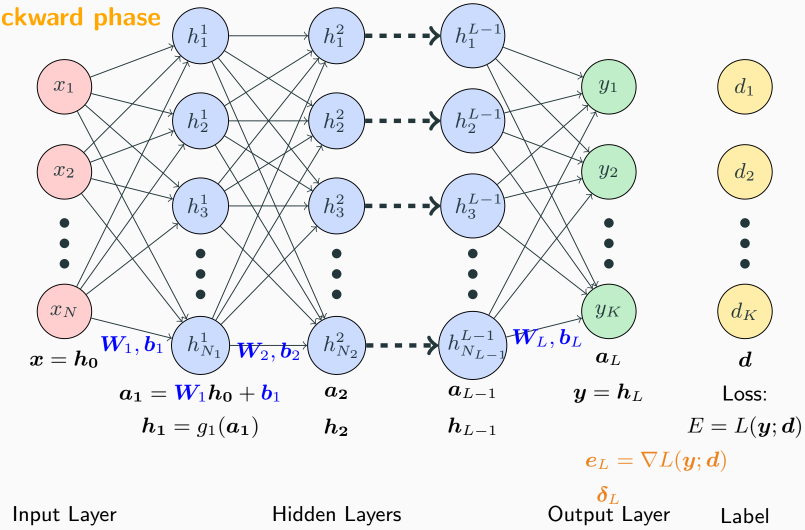


## Error backpropagation

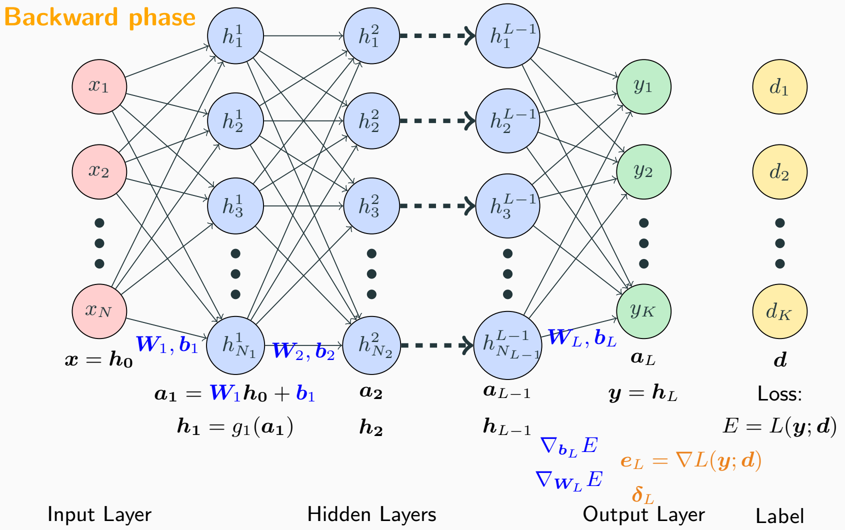


## Error backpropagation

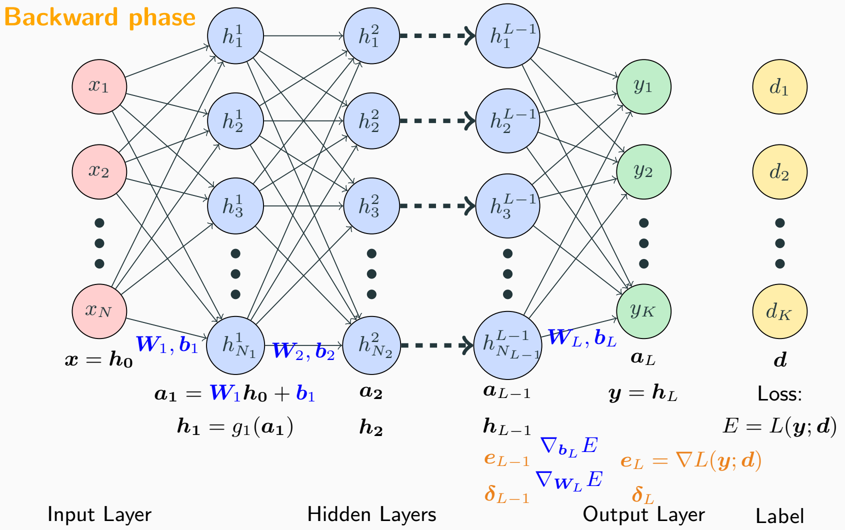
**Backward phase**



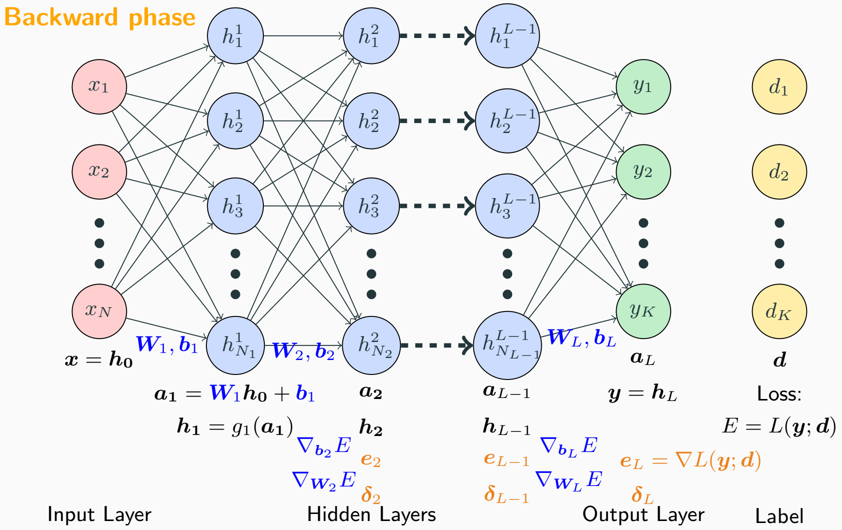
## Error backpropagation



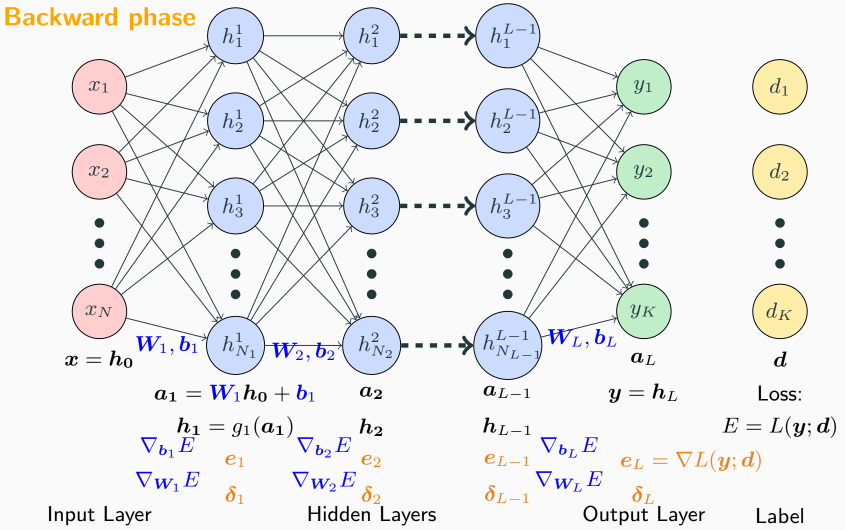
## Error backpropagation



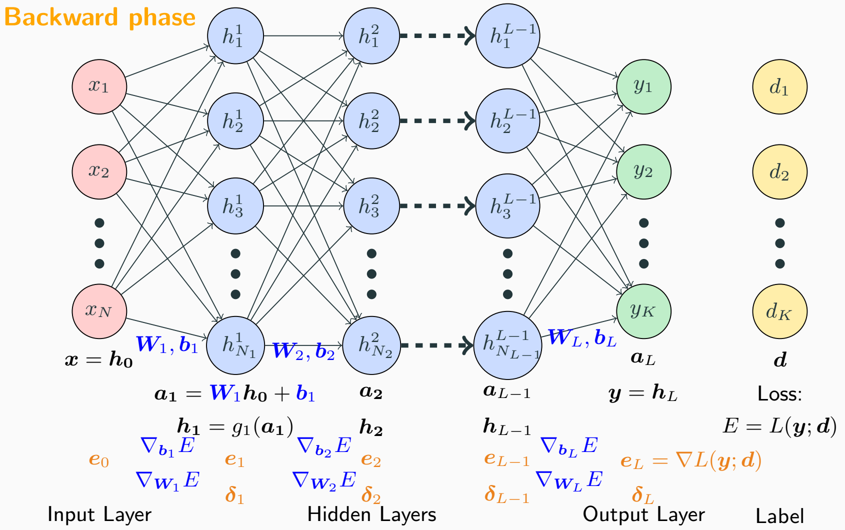
## Error backpropagation



## Error backpropagation



## Error backpropagation



## Error backpropagation in practice

**Training loss:**

$$E(\boldsymbol{W}) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{T}} L(\boldsymbol{y}^i; \boldsymbol{d}^i)$$

- The backpropagation procedure computes $\nabla_{\boldsymbol{W}} L(\boldsymbol{y}^i; \boldsymbol{d}^i) = \nabla_{\boldsymbol{W}} L(f(\boldsymbol{x}^i; \boldsymbol{W}); \boldsymbol{d}^i)$.
- This has to be done for each data point $\boldsymbol{x}^i \in \mathcal{T}$.
- By linearity, the final gradient $\nabla E(\boldsymbol{W})$ is the sum of all individual gradients $\nabla_{\boldsymbol{W}} L(\boldsymbol{y}^i; \boldsymbol{d}^i)$.
- These gradients are summed sequentially (no need to store each individual gradients).
- In general we do not compute the exact gradient...

## Error backpropagation in practice

**Batch loss:**

$$E(\boldsymbol{W}) \approx \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{S}} L(\boldsymbol{y}^i; \boldsymbol{d}^i), \quad \text{with} \quad \mathcal{S} \subset \mathcal{T}$$

- The backpropagation has to be done for each visited data point $\boldsymbol{x}^i \in \mathcal{S}$ of the batch.
- The gradient for each point $\boldsymbol{x}^i$ is added to the running gradient = current gradient estimation.
- Once the noisy estimated gradient is used as a gradient step, one needs to set the gradients to zero: See PyTorch `torch.zero_grad()` procedure.

## Why is the backpropagation so efficient ?

- Avoid to re-do numerous computations
- No parameters need to be tuned
- Easy to implement
- This method can be seen as a Jacobian multiplication "in the right direction"

Some limitations :

- Memory consuming
- Backpropagation learning does not require normalization of input vectors; however, normalization could improve performance.

# Questions?

**Sources, images courtesy and acknowledgment**

Charles Deledalle | V. Lepetit | L. Masuch